Improving the Precision of Type Inference Algorithms with Lightweight Heuristics

Publication Summary

Nevena Milojković
University of Bern
Bern, Switzerland
nevena@inf.unibe.ch

Abstract

Dynamically-typed languages allow faster software development by not posing the type constraints. Static type information facilitates program comprehension and software maintenance. Type inference algorithms attempt to reconstruct the type information from the code, yet they suffer from the problem of false positives or false negatives. The use of complex type inference algorithms is questionable during the development phase, due to their performance costs. Instead, we propose lightweight heuristics to improve simple type inference algorithms and, at the same time, preserve their swiftness.

1 Introduction

Static type information is irreplaceable when it comes to program comprehension and software maintenance [HKR+14, SH14]. Not only does software maintenance require less time in statically-typed languages, but already static type information without static type checking has a positive impact on program comprehension [SH14]. While in statically-typed languages a developer may use available static type information to reason about software [SMDV08], in dynamically-typed languages she is often forced to exercise the code to test type hypotheses [KBR14]. On the other hand, dynamically-typed languages offer developers more expressiveness during the development phase [MHR+12] and, consequently, reduce development time [Han10].

In order to bring together the two worlds, numerous type inference techniques have been developed in the last several decades. Some of them employ solely statically collected information, while the others depend on the program execution. Some of them depend on typing the whole program, while the others analyse an isolated expression. Program execution offers precise type information, but it gives only the narrow scope of all the possibilities [PTP07]. Static analysis is an NP-hard problem [Sus97, Lan92], hence it suffers from *false positives or negatives*, *i.e.,* it indicates as a result something that cannot happen at run-time, or misses some results.

These algorithms need to be fast, in order not to break the development flow. However, it is intuitive that algorithms that employ more information and perform flow analysis are expected to be more precise. Hence, in order to be usable by a developer they need to trade precision for speed.

We argue that lightweight heuristics may be employed to improve the accuracy of control-flow and data-flow insensitive type inference algorithms, that is, to mitigate the number of false positives or negatives, without a significant loss of speed. These heuristics are founded on the information that is easily accessible from the source code either statically or at run time. By performing lightweight code analysis, we believe that it is possible to augment the precision of a simple type inference algorithm, while preserving its simplicity and swiftness. Hence, they would remain fast and practical during coding tasks, without breaking work flow.

We have used two simple type inference algorithms as basic algorithms: *RoelTyper* [PMW09] and the *Cartesian Product Algorithm (CPA)* [Age95], on top of which we have implemented four lightweight heuristics.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org).
07-09 June 2017, Madrid, Spain.

Two heuristics employ static class usage frequency information [MN16], one heuristic employs the information from inline caches [MBGN16], and one uses type hints in method argument names [MGN17]. They are all described in Section 2. We discuss open questions in Section 3.

2 Lightweight heuristics

The simplest approach to infer types for a variable is to track down the assignments and set of messages sent to the variable, without any flow analysis [PMW09]. The reconstructed types for a variable are represented by the union of the types assigned to the variable and the types that understand all the messages sent to the variable. It is very fast, but it provides precise results for a bit less then a 60% of variables. Its main obstacles are polymorphism and cross-hierarchy polymorphism usage. When the interface of a variable consists of popular selectors, implemented in multiple independent hierarchies, this approach suffers from *false positives, i.e.,* classes that understand the interface of the variable, but do not represent its run-time type.

Instead of complicating the design of the algorithm, as it would increase the time needed for the analysis, we propose ordering of the resulting classes. As each of the classes inferred as possible types for a variable is more or less likely to be correct, we explore possible heuristics for their sorting. Naturally, an object that exists at run time needs to be created at some point during program execution. This can be achieved in any of the following ways:

1. invoking a constructor
2. invoking a (factory) method that plays the role of a constructor, but is not a constructor per se
3. via reflection

Our hypothesis is that the more frequently the class is instantiated in the code, the more likely it will represent a type of a variable at run time. Since this can be done in three different ways, we propose three heuristics implemented on top of RoelTyper to mitigate the number of false positives.

On the other hand, in the presence of reflection, or dynamic class loading [LSS+15], static type inference algorithms miss certain types, hence they lose type information. This means that they suffer from the problem of *false negatives, i.e.,* they omit from the results the classes that represent a variable type at run time. We propose the fourth heuristic built on top of CPA. The Cartesian Product Algorithm attempts to reconstruct the type flow through the assignments and method calls, and it infers the return type of the method based on the Cartesian product of the argument types. The heuristic uses type hints encoded in the method argument names to improve the algorithm's precision in the presence of reflection.

We measured the time needed to provide a type feedback to assess whether these heuristics are fast enough to be used for program comprehension. The introduced overhead is less than 5% in case of RoelTyper and about 10% in case of CPA, which we deem acceptable. The following subsections present three heuristics for ordering classes, and the heuristic used to recover missed types, respectively.

2.1 Heuristic based on the class instantiation frequency

Regardless of how complicated the control flow of a piece of software is, all the instances to which the variable may be bound at run time must be created somewhere in the code. The usual way is to invoke a constructor of the desired class, leading us to suppose that the more frequently the class is instantiated throughout the code, the more likely it is that it will represent a variable type at run time. Hence, we propose ordering the inferred classes for a variable based on the frequency of constructor calls for a class. We have implemented a prototype for Pharo Smalltalk[1] and used it to evaluate the approach. The heuristic showed a more than twofold improvement when compared with the basic approach.

2.2 Heuristic based on the class name occurrence frequency

While an instance of a class may be created by explicitly invoking a constructor of the desired class, it may also be created by the use of a factory method. Some languages do not pose restrictions on constructor naming, thus it may be difficult to statically track all constructor invocations. Any method may play the role of a constructor. These factory methods may directly invoke a constructor and, in this case, it is easy to identify them. But, they may also invoke some other factory method that will, in return, invoke a constructor and, in this case, more precise static analysis should be performed in order to avoid false positives. Also, if a factory method contains a message send `self basicNew`, it is costly, if at all possible, to statically determine whether an object of the corresponding class, or any of its subclasses will be created.

We propose also to explore a heuristic of ordering possible types for a variable based on the frequency of class name occurrence in source code, rather than on the class instantiation frequency. We have used for the evaluation the implemented prototype. Interestingly, this heuristic showed slightly better results than the previous one.

[1] `http://www.pharo.org`

2.3 Heuristic based on the class frequency from inline caches

Recent studies show that reflective features are quite frequently used in dynamically-typed languages [HH09, RLBV10, CRTR13]. Due to dynamic class loading or high use of reflection, static analysis can miss the use of certain types [LSS+15]. This imposes difficulties for static analysis, as sometimes it cannot be known at compile time which class will be instantiated or even created.

In the presence of dynamic binding, many virtual machines for dynamic languages employ Just-in-Time compilers to speed up the execution [HCU91]. These compilers use inline caches, which locally store the information about methods previously executed for a message send. Beside method information, these caches also contain the information about the type of the receiver for a message send.

We hypothesise that the class usage frequency as a type of the receiver at run-time, read from the inline caches, may serve as a reliable proxy for the likelihood of a variable being of a certain type. This information can be used to order statically-inferred types. As runtime information is easily accessible from the virtual machine, no instrumentation is required.

The evaluation showed results very similar to those of the heuristics based on purely static information.

2.4 Heuristic based on the type hints from method argument names

In the presence of reflection, or dynamic class loading, static type inference algorithms may underapproximate possible types for a variable.

On the other hand, in order to partially compensate for the lack of static type information, a common idiom in dynamically-typed languages is to provide a type annotation for method arguments [Bec97, Zan13, Bol10]. These annotations are mainly intended to improve program comprehension, but they are also used as an input for some development tools, *e.g.,* code completion, in order to improve the results.

We hypothesise that these annotations from method argument names may be employed to improve the precision of a type inference in cases where the type of the variable cannot be statically inferred by traditional approaches. We propose a heuristic to augment a type inference algorithm whose precision significantly depends on the correctly inferred types for method arguments [Age95].

We have implemented a prototype used for evaluation. The obtained results show that the augmented algorithm outperforms the basic one significantly.

3 Future work and open questions

We chose RoelTyper and CPA, as they represent simple and swift type inference algorithms. While we find them representative for the field, there is an open question whether augmenting other type inference algorithms would yield different results.

Our idea was to start with the easiest heuristics to calculate, in order to retain the speed of the underlying algorithms. Our choice of the heuristics fell on the four heuristics we have presented, as they are simple enough not to complicate the type inference process. For instance, the heuristic that we employed on top of CPA provides results in only 10% overhead, and the other three heuristic introduce an overhead of at most 5%.

We list here some possible directions of research:

- We considered all classes from the Smalltalk libraries as available for type inference, *i.e.,* that any of the classes may represent a type of a variable. Not every one of these library classes is reachable (through the call graph) from the project under analysis. A first step would be to construct the conservative call graph starting from the analysed project, and restrict a set of types only to actually reachable classes. One of the problems in the call graph construction comes from the use of reflective and dynamic features.

- The Smalltalk community heavily uses type predicates [CRT+14], whose usual name consists of the present tense of the verb "to be" plus the camel case notation of the expected type name, *e.g.,*isCircle. Extraction of the expected class name from the predicates may reveal the popular classes. Each class may be given a *score* based on its popularity, and these scores may be used for class ordering.

- Reflective features are also commonly used in many dynamically-typed languages, especially dynamic method invocation [CRTR13], and for some of these features it is possible to statically determine the exact name of the invoked method, *e.g.,*object.perform("size") where the message size is sent to the object. By extracting these string values, one may assume that object is supposed to be some kind of collection. This may improve the results in any of the type inference algorithms.

- Developers may encode type information into the comments and documentation, hence their analysis in the search of class names may reveal the popular classes. For example, one of the method comments in Glamour[2] states: "Answers a string

[2] `http://www.smalltalkhub.com/#!/~Moose/Glamour`

explaining shortcut." Its analysis would reveal the use of String type.

- In the case of false positives, classes may also be sorted based on their distance from the classes that contain the analysed variable. For example, possible types that belong to the same package may be prioritised, next would be the group of classes from the same project, but a different package, and at the end the remaining classes.

- Each class (*e.g., A*) may contain a *score* of every other class (*e.g., B*) that is used inside it. That is, if there is an explicit mention of class name B within the class A, or if B represents a type of the variable defined by the class A. This could represent the closeness of the two classes.

In the end, we believe that it is important to explore various heuristics, especially those that accommodate language idioms, in order to improve static type analysis, without introducing significant overhead.

Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

References

[Age95] Ole Agesen. The Cartesian product algorithm. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 2–26, Aarhus, Denmark, August 1995. Springer-Verlag.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[Bol10] M. Bolin. *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript*. O'Reilly Media, 2010.

[CRT+14] Oscar Callaú, Romain Robbes, Éric Tanter, David Röthlisberger, and Alexandre Bergel. On the use of type predicates in object-oriented software: The case of Smalltalk. In *Proceedings of the 10th ACM Dynamic Languages Symposium (DLS 2014)*, pages 135–146, Portland, OR, USA, 2014. ACM Press.

[CRTR13] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. *Empirical Software Engineering*, 2013.

[Han10] Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. *SIGPLAN Not.*, 45(10):22–35, October 2010.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Ecoop'91 european conference on object-oriented programming: Geneva, switzerland, july 15–19, 1991 proceedings. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[HH09] Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, pages 19–28. Australian Computer Society, Inc., 2009.

[HKR+14] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[KBR14] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Asking and answering questions during a programming change task in the Pharo language. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14, pages 1–11, New York, NY, USA, 2014. ACM.

[Lan92] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[MBGN16] Nevena Milojković, Clément Béra, Mohammad Ghafari, and Oscar Nierstrasz. Inferring types by mining class usage frequency from inline caches. In *Proceedings of International Workshop on Smalltalk*

Technologies (IWST 2016), pages 6:1–6:11, 2016.

[MGN17] Nevena Milojković, Mohammad Ghafari, and Oscar Nierstrasz. Exploiting type hints in method argument names to improve lightweight type inference. In *25th IEEE International Conference on Program Comprehension*, 2017.

[MHR+12] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. *SIGPLAN Not.*, 47(10):683–702, October 2012.

[MN16] Nevena Milojković and Oscar Nierstrasz. Exploring cheap type inference heuristics in dynamically typed languages. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016*, pages 43–56, New York, NY, USA, 2016. ACM.

[PMW09] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th Symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[PTP07] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, 42(10):535–552, 2007.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. *SIGPLAN Not.*, 45(6):1–12, June 2010.

[SH14] Samuel Spiza and Stefan Hanenberg. Type names without static type checking already improve the usability of APIs (as long as the type names are correct): An empirical study. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 99–108, New York, NY, USA, 2014. ACM.

[SMDV08] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34:434–451, July 2008.

[Sus97] Horwitz Susan. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[Zan13] Matt Zandstra. *PHP Objects, Patterns, and Practice*. Apress, Berkely, CA, USA, 4th edition, 2013.